**Data Structures and Algorithm – Day 2 –March -2021**

**Assignment 2 :**

**Find the time complexity of the following Pseudo Code**

**for( i = 1 ; i <= n ; i = i\*2)**

**for (j=1;j<=I;j++)**

**print(“Hello”)**

**Output : if n=3**

**Hello**

**Hello**

**Hello**

**int i=0;** This will be executed only **once**. The time is actually calculated to i=0 and not the declaration.

**i < N;** This will be executed **N+1** times

**i++ ;** This will be executed **N** times

So the number of operations required by this loop are

**{1+(N+1)+N} = 2N+2**

Note: This still may be wrong, as I am not confident about my understanding on calculating time complexity

**What I want to know ?**

Ok, so these small basic calculations I think I know, but in most cases I have seen the time complexity as

**O(N), O(n2), O(log n), O(n!)**.... and many [other](http://en.wikipedia.org/wiki/Time_complexity),

You add up how many machine instructions it will execute as a function of the size of its input, and then simplify the expression to the largest (when N is very large) term and can include any simplifying constant factor.

We are interested in the performance of the algorithm as N becomes large.

Consider the two terms 2N and 2.

What is the relative influence of these two terms as N becomes large? Suppose N is a million.

Then the first term is 2 million and the second term is only 2.

For this reason, we drop all but the largest terms for large N.

So, now we have gone from 2N + 2 to 2N.

Traditionally, we are only interested in performance *up to constant factors*.

This means that we don't really care if there is some constant multiple of difference in performance when N is large. The unit of 2N is not well-defined in the first place anyway. So we can multiply or divide by a constant factor to get to the simplest expression.